chapter

6 非同步設計

學習目標

- 區別同步與非同步
- 運用 Promise API
- 善用 async、await
- 認識 for-await-of
- 使用非同步產生器函式

6.1 初識非同步

到目前為止介紹過的範例都是單一流程，也就是執行.js 從開始至結束只有一個主流程；JavaScript 最初為瀏覽器而生，而使用者在操作瀏覽器時會有各種操作，若想在某操作事件發生時執行對應的程式流程，該怎麼做呢？這就涉及了非同步設計的領域了，而非同步是 JavaScript 的天性，就算離開瀏覽器的世界，在 Node.js 等後端中使用 JavaScript，也經常要處理非同步議題，因而也是 JavaScript 開發者必須掌握的重點之一。

6.1.1 使用 `setTimeout()`

從頭至尾只使用一個主流程設計有什麼問題嗎？來看個例子吧！如果要設計一個龜兔賽跑遊戲，賽程長度為 10 步，每經過一秒，烏龜會前進一步，兔子可能前進兩步或睡覺，那該怎麼設計呢？如果用目前學過的流程語法來說，可能會如下設計：

async-callback tortoise-hare.js

```
const INTERVAL = 1000;
const TOTAL_STEP = 10;

let tortoiseStep = 0;
let hareStep = 0;

console.log('龜兔賽跑開始...');
do {
    let time = Date.now();
    while(Date.now() - time < INTERVAL) {} ◀── ❶計時一秒

    tortoiseStep += 1; ◀── ❷烏龜走一步
    let running = Math.ceil(Math.random() * 10) % 2; ◀── ❸隨機決定是否前進
    if(running) {
        hareStep += 2; ◀── ❹兔子走兩步
    }
    else {
        console.log('兔子睡著了 zzzz');
    }

    if(tortoiseStep < TOTAL_STEP) {
        console.log(`烏龜跑了 ${tortoiseStep}  步...`);
    }
    else {
        console.log('烏龜抵達終點！');
```

```
    }

    if(hareStep < TOTAL_STEP) {
        console.log(`兔子跑了 ${hareStep}  步...`);
    }
    else {
        console.log('兔子抵達終點！');
    }
} while(tortoiseStep < TOTAL_STEP && hareStep < TOTAL_STEP);
```

目前程式只有一個流程，就是從.js 檔案開始至結束的流程，Date.now()可以取得系統時間 1970 年 1 月 1 至今經過的毫秒數，結合迴圈重複判斷，以實現每秒一次的需求❶。tortoiseStep 遞增 1 表示烏龜走一步❷；Math.random()會隨機產生 0 到 1 間小數，乘以 10 使用 Math.ceil()捨去小數後，取 2 的餘除就有 0 或 1 的結果，非 0 值在成立與否的情境，像是 if 判斷時會被作為成立，也就是兔子會是隨機前進❸，如果前進會將 hareStep 遞增 2，表示兔子走兩步❹，烏龜或兔子其中一個走完 10 步就離開迴圈，表示比賽結束。

由於程式只有一個主流程，就不得不將計時任務，以及烏龜與兔子的行為混雜在同一流程中撰寫，如果可以分別定義計時任務以及烏龜、兔子的流程，程式邏輯會比較清楚。

setTimeout()函式雖然不是 ECMAScript 本身的標準，然而在瀏覽器環境或 Node.js 都支援，可以用來實現計時任務，而烏龜、兔子的行為，可以定義在不同的函式，並在逾時（Timeout）事件發生時呼叫。例如：

async-callback tortoise-hare2.js

```
const INTERVAL = 1000;
const TOTAL_STEP = 10;

function tortoise(step = 1) {    ←— ❶烏龜的流程
    if(step < TOTAL_STEP) {
        console.log(`烏龜跑了 ${step} 步...`);
        setTimeout(tortoise, INTERVAL, step + 1); ←— ❷一秒後執行
    } else {
        console.log('烏龜抵達終點！');
    }
}

function hare(step = 2) {    ←— ❸兔子的流程
    if(step < TOTAL_STEP) {
        let running = Math.ceil(Math.random() * 10) % 2;
        if(running) {
```

```
            console.log(`兔子跑了 ${step} 步...`);
            setTimeout(hare, INTERVAL, step + 2); ◀── ❹一秒後執行
        }
        else {
            console.log('兔子睡著了 zzzz');
            setTimeout(hare, INTERVAL, step);
        }
    }
    else {
        console.log('兔子抵達終點！');
    }
}

console.log('龜兔賽跑開始...');
setTimeout(tortoise, INTERVAL);  ◀── ❺一秒後執行 tortoise 函式
setTimeout(hare, INTERVAL);      ◀── ❻一秒後執行 hare 函式
```

在這個範例中，使用 tortoise 與 hare 函式分別定義了烏龜與兔子的流程❶❸，計時任務是由 setTimeout()函式負責，它的第一個參數接受要執行的函式，第二個參數是毫秒指定的時間（若不指定，預設為 0 毫秒），之後的參數可以是呼叫指定的函式提供之引數；首次執行 tortoise 與 hare 函式的時間是一秒之後❺❻，之後若還沒抵達終點，就再次呼叫 setTimeout()函式❷❹。執行時的結果會像是…

```
兔賽跑開始...
烏龜跑了 1 步...
兔子睡著了 zzzz
烏龜跑了 2 步...
兔子睡著了 zzzz
烏龜跑了 3 步...
兔子跑了 2 步...
烏龜跑了 4 步...
兔子跑了 4 步...
烏龜跑了 5 步...
兔子睡著了 zzzz
烏龜跑了 6 步...
兔子睡著了 zzzz
烏龜跑了 7 步...
兔子睡著了 zzzz
烏龜跑了 8 步...
兔子跑了 6 步...
烏龜跑了 9 步...
兔子跑了 8 步...
烏龜抵達終點！
兔子抵達終點！
```

6.1.2　同步？非同步？

電腦科學領域有許多名詞，不過多半沒有嚴謹的定義；然而，既然這邊要討論非同步，總還是得定義一下非同步與同步（Synchronous），以便後續進行討論。

在本章之前的程式碼或範例，都有個明顯的特徵，**無論是運算式、陳述句或是函式，都是在定義的任務完成之後，才會往下一個運算式、陳述句或函式執行，這樣的流程稱為同步**。例如，若已知 foo()函式的任務為顯示 foo 字樣，若底下的程式碼執行結果，是依序顯示 begin、foo、end，那麼這個程式碼片段就是同步的流程：

```
console.log('begin');
foo();
console.log('end');
```

在本章之前定義過的函式，都是這類函式，姑且稱為同步函式，當然，要實作這樣的 foo()函式並不難，例如：

```
function foo() {
    console.log('foo');
}
```

如果是另一種 foo()函式實作，會令方才的程式片段顯示 begin、end、foo 呢？也就是 foo()的任務未完成前，就執行了後續的程式流程，那麼這樣的流程就是非同步的，而這類函式就被稱為非同步函式。

在先前的 tortoise-hare2.js 中使用了 setTimeout()，可以指定的時間後才執行指定的函式；若拿 setTimeout()來設計底下的 foo()函式，就可以達到非同步的需求。

async-callback async-foo.js

```
// 非同步函式
function foo() {
    setTimeout(console.log, 1000, 'foo');
}

// 非同步流程，執行後顯示 begin、end、foo
console.log('begin');
foo();
console.log('end');
```

實際上，計時本身就是個獨立於主流程的任務，逾時是個事件，在事件發生時必須執行某個指定操作，像這類**獨立於程式主流程的任務、事件生成，以及處理事件的方式，稱為非同步（Asynchronous）**。

在瀏覽器的環境時，獨立於主流程的任務可能是下載檔案，下載完成是個事件，有事件發生時必須執行某個指定操作；在 Node.js 的環境中，獨立於主流程的任務可能是寫入檔案，寫入完成是個事件，有事件發生時必須執行某個指定操作；為了滿足諸如此類的需求，非同步根本上就是與 JavaScript 息息相關的議題。

如果你曾經學習過其他具有多執行緒（Multi-thread）特性的語言（例如 Java），可能會想到，為什麼不透過多執行緒呢？確實地，在支援多執行緒的語言中，執行緒是用來實現非同步的方式之一；然而，**JavaScript 本身不支援多執行緒，而且 JavaScript 引擎是以單執行緒方式執行程式碼**。

對於熟悉多執行緒的許多開發者來說，單執行緒可以實現非同步，乍聽之下難以想像，實際上，**JavaScript 執行環境會在事件迴圈（Event loop），不斷地檢查事件佇列（Event queue），當事件發生時，並不是馬上執行指定的函式，而是將事件排入佇列，在迴圈下一輪的檢查時，才將佇列中事件對應的任務依序執行完成**。

以 async-foo.js 為例，`foo()`執行時，`setTimeout()`會令瀏覽器或 Node.js 進行計時，然後 `setTimeout()`就執行完畢了，這也意謂著 `foo()`函式執行完畢，因此才會往下一行 `console.log('end')`執行，當逾時事件發生時，`setTimeout()`指定的函式會排入事件佇列，在迴圈下一輪檢查佇列時執行。

簡而言之，JavaScript 是以單執行緒，不斷地在事件迴圈中檢查事件佇列，若佇列中有任務的話就依序執行完成；這就意謂著，JavaScript 沒有多執行緒切換的機制，也就是說，**如果某個事件對應的任務執行過久，佇列中後續事件對應的任務就會被卡住**，若是在使用者操作瀏覽器的時候發生了這類情況，就會感覺操作畫面被凍結，或者是發生無回應的狀態；這也表示，在支援多執行緒的語言中 sleep 之類的函式或方法，在 JavaScript 中無法實現。

提示 »» 在 HTML5 規範包含 Web Worker API，可以提供多執行緒，然而這個支援是由「瀏覽器」提供，而不是 JavaScript 引擎。

6.1.3　非同步與回呼

對於同步函式，任務執行完之後會傳回結果；然而非同步函式呼叫時，被指定的任務未完成前，函式就會立即返回，這麼一來，該怎麼取得任務執行結果呢？

對於非同步函式想要取得執行結果，方式之一是使用回呼函式。例如，若 asyncFoo()函式會在兩秒之後，使用指定的引數乘上隨機產生的數字，若要取得該結果的話，可以如下設計：

async-callback async-foo2.js

```
function asyncFoo(n, callback) {
    setTimeout(
        () => callback(n * Math.random()),
        2000
    );
}

asyncFoo(10, console.log);
```

這個範例在執行完 setTimeout()後，asyncFoo()函式立即返回，而在兩秒之後完成任務，以結果呼叫了指定的 callback 函式，這種回呼模式直覺而簡單，一些基礎程式庫，像是 Node.js 中想要讀取檔案，可以使用 fs 模組的 readFile()函式，若想取得檔案讀取結果並做進一步處理，就必須指定回呼函式。例如：

```
let fs = require('fs');
fs.readFile('files.txt', function(_, files) {
    console.log(files.toString());
});
```

提示 »» 這邊談到的模組，是指 Node.js 本身的模組，並非 ECMAScript 規範的模組，Node.js 也支援 ECMAScript 模組，ECMAScript 模組將在第 10 章進行討論。

在事情簡單時，回呼模式沒什麼問題，然而，若希望任務完成後，接續地執行下一個非同步任務時，就會發生回呼函式形成巢狀結構的問題。例如：

```
let fs = require('fs');
fs.readFile('files.txt', function(_, files) {
    files.toString().split('\r\n').forEach(function(file) {
        fs.readFile(file, function(_, content) {
            console.log(content.toString());
```

```
        });
    });
});
```

在 JavaScript 的應用中，非同步地執行任務是常見需求，回呼函式形成巢狀結構的問題可能會變得嚴重，因而引發**回呼地獄（Callback hell）**的問題，令撰寫程式或後續閱讀程式碼發生困難。

提示 »» 在方才的範例中出現了 `require()`，這是 Node.js 載入模組時的函式，本書不談 Node.js，然而在第 10 章有談到模組管理，其中略為解釋了一下 Node.js 的 `require()`；為了將焦點集中在理解非同步本身，後續範例會使用 **`setTimeout()`** 來代表獨立於主流程的某個非同步任務。

例如若使用 async-foo2.js 的 `asyncFoo()` 為例，想在取得任務結果之後，再次用來呼叫 `asyncFoo()`，連續三次地呼叫之後，就會寫成這樣如下結構：

```
asyncFoo(10, r1 => {
    asyncFoo(r1, r2 => {
        asyncFoo(r2, r3 => {
            console.log(r3);
        });
    });
});
```

可以看到的，就算使用了 ES6 箭號函式，在這種需求之下，會因為巢狀的回呼函式，造成可讀性迅速下降。

回呼模式不是不好，若非同步操作取得結果後，不會有進一步執行非同步任務的需求來說，回呼模式直覺而簡單；然而，若發現需求已經超出了原本預想的範圍，程式碼開始有形成回呼地獄的傾向時，就應該適時重構，以免影響程式的可讀性，重構時可以採取的方向之一是 Promise 模式，而 ES6 以後，提供了 `Promise` API 來支援此模式，而這是下一節要探討的內容。

提示 »» 非同步操作有多種模式，回呼或 Promise 只是其中兩種，有興趣認識更多模式的話，可以參考〈非同步操作的多種模式[1]〉。

[1] 非同步操作的多種模式：openhome.cc/Gossip/Programmer/AsyncPatterns.html

6.2 Promise

在電腦科學領域中，名詞通常沒有嚴謹的定義，Promise 模式在不同語言中會有不同的稱呼，也有人稱為 Future、Delay 或 Deferred 模式，由於 ES6 提供了 Promise API，在 JavaScript 領域也就習慣用 Promise 模式來稱呼，接下來這一節，就是要直接從 Promise API 認識 Promise 模式。

6.2.1 Promise 實例

在 ES6 以後，**Promise 實例可以作是非同步函式的傳回值，正如名稱暗示的，Promise 實例代表著「承諾」在未來提供任務執行結果。**以 6.1.3 的 async-foo2.js 為例，可以使用 Promise 來實作 asyncFoo()函式，令其傳回 Promise 實例：

promise async-foo.js

```
function asyncFoo(n) {
    return new Promise(resolve => {
        setTimeout(
            () => resolve(n * Math.random()), ◄── ❶ 任務達成
            2000
        );
    });
}

asyncFoo(10)
    .then(r1 => asyncFoo(r1))  ◄── ❷ 循序風格的呼叫方式
    .then(r2 => asyncFoo(r2))
    .then(r3 => console.log(r3));
```

Promise 實例建立後，處於**未定（Pending）狀態**，在建立 Promise 實例時可以指定回呼函式，該函式可以具有參數，第一個參數慣例上常命名為**resolve**，這兩個參數會各自接受函式，這邊僅定義了 resolve 參數，若呼叫 resolve()函式，表示 Promise 指定的任務**達成（Fulfilled）狀態**，呼叫 resolve()時指定的引數，就是任務達成的結果❶。

Promise 實例具有 then()方法，then()接受的回呼函式可以具有參數，若 Promise 指定的任務達成，那麼呼叫 resolve()時指定的引數，就會是 then()回呼函式的參數值，回呼函式可以傳回任何值，通常會是 Promise 實例或是 thenable 物件（具有 then()方法的物件），若是後兩者，代表下個待達成的任

務，then()方法會傳回 Promise，因此可以形成方法鏈進行連續呼叫❷，各個then()呼叫的回呼函式中，可以取得任務達成結果。

就程式碼閱讀來說，Promise 的撰寫風格就像是循序的，也避免了回呼地獄的問題，不過順序的保證，僅限於 then()指定的任務，上面的範例若在最後一行加上了 console.log('end')：

```
…略
asyncFoo(10)
    .then(r1 => asyncFoo(r1))
    .then(r2 => asyncFoo(r2))
    .then(r3 => console.log(r3));
console.log('end');
```

那麼會先顯示 end 字樣，後續才會顯示非同步任務的結果，如果想要顯示非同步任務的結果，最後才顯示 end 字樣，方式之一是再寫個 then()：

```
…略
asyncFoo(10)
    .then(r1 => asyncFoo(r1))
    .then(r2 => asyncFoo(r2))
    .then(r3 => console.log(r3))
    .then(_ => console.log('end'));
```

另一個方式是使用ES8的async、await機制，這會在6.3再來討論。Promise的then()方法，也可以直接接受 Promise 實例，若不在乎前一個 Promise 的結果，只需要在前一個 Promise 完成後執行時使用。例如：

```
asyncFoo(10)
    .then(asyncFoo(20));
```

在建立 Promise 實例時指定的回呼函式，可以定義第二個參數，慣例上常命名為 **reject**，如果指定的任務無法達成，可以使用傳入的 reject 函式來否決任務，此時 Promise 就會處於**否決（Rejected）狀態**。例如：

promise resolve-reject.js

```
function randomDivided(divisor) {
    return new Promise((resolve, reject) => {
        let n = Math.floor(Math.random() * 10);
        if(n !== 0) {
            resolve(divisor / n);
        } else {
            reject('Shit happens: divided by zero');
        }
    });
```

```
}

randomDivided(10)
    .then(
        n   => console.log(n),
        err => console.log(err)
    );
```

在上例中，若 n 不為 0 任務就會達成，若為 0 就會否決任務，若有定義 then() 的第二個回呼函式，該函式就會被呼叫，reject() 接受的引數，可以成為 then() 第二個回呼函式的引數，在這邊 reject() 時指定了字串值。

本質上，Promise 被否決時，表示任務因為某個錯誤而無法執行下去，如果想要突顯錯誤處理的流程，建議使用 catch() 方法。例如：

```
randomDivided(10)
    .then(n   => console.log(n))
    .catch(err   => console.log(err));
```

會使用 catch 這個字眼，其實與 JavaScript 錯誤處理語法 try-catch-finally 有關，錯誤處理在 JavaScript 中是另一個重要課題，將在第 7 章時再來討論。

6.2.2 銜接 Promise

Promise 建構式本身擁有幾個特性，也就是以 Promise 為名稱空間的函式（不是定義在 Promise.prototype 上的特性），主要目的是作為 API 的銜接。例如，方才的 resolve-reject.js 中，除了直接以 new 建構來銜接 Promise API，也可以這麼撰寫：

promise resolve-reject2.js

```
function randomDivided(divisor) {
    let n = Math.floor(Math.random() * 10);
    if(n !== 0) {
        return Promise.resolve(divisor / n);
    } else {
        return Promise.reject('Shit happens: divided by zero');
    }
}

randomDivided(10)
    .then(
        n   => console.log(n),
        err => console.log(err)
    );
```

Promise.resolve()可以接受任何值，包括 Promise 實例或是 thenable 物件，這些物件任務達成的值，會成為 Promise.resolve()傳回 Promise 實例的達成值；**Promise.reject()**也可以接受任何值。

如果手邊有多個 Promise 實例，而且不在意達成的順序，只要達成的結果，是依照指定的 Promise 順序排列就可以的話，可以使用 **Promise.all()**，它接受 Promise 組成的可迭代物件，並傳回 Promise 實例，例如：

```
Promise.all([randomDivided(10), randomDivided(20), randomDivided(30)])
       .then(
           results => console.log(results[0], results[1], results[2]),
           err => console.log(err)
       );
```

Promise.all()就像是將一組 Promise 組合為一個 Promise，就上例來說，如果三個 Promise 的任務都達成了，那麼 results 就會是三個任務的結果陣列，元素是依照 Promise.all()最初指定的 Promise 順序；然而，如果有任何一個 Promise 被否決了，無論是否還有其他 Promise 的任務在執行，都會直接呼叫 then()指定的第二個回呼函式，未達成的 Promise 還是會繼續執行，也無法取得其他有達成任務的 Promise 之結果。

Promise.race()函式則接受 Promise 組成的可迭代物件，這組 Promise 任一個先達成或否決的話，就會當成 Promise.race()傳回的 Promise 達成或否決的結果，不過其他 Promise 還是會繼續執行就是了。一個使用 Promise.race()的例子是：

```
Promise.race([randomDivided(10), randomDivided(20), randomDivided(30)])
       .then(
           result => console.log(result),
           err => console.log(err)
       );
```

那麼，如果想指定一組 Promise，並且在全部的 Promise 達成或否決之後，再來判斷各個 Promise 的狀態，該怎麼做呢？這個必須自行設計來達成。例如，可以設計一個 allSettled()函式：

promise resolve-reject3.js

```
function randomDivided(divisor) {
    let n = Math.floor(Math.random() * 10);
    if(n !== 0) {
        return Promise.resolve(divisor / n);
```

```
    } else {
        return Promise.reject('Shit happens: divided by zero');
    }
}

function allSettled(promises) {
    function statusObject(promise) {  ◀── ❶轉換為狀態物件
        return promise.then(
            v => ({status: 'fulfilled', value: v }),
            err => ({ status: 'rejected', reason: err})
        );
    }
    return Promise.all(promises.map(statusObject)); ◀── ❷收集狀態物件
}
                                                    ❸判斷狀態物件
allSettled([randomDivided(10), randomDivided(20), randomDivided(30)]) ↓
    .then(statusObjs => statusObjs.filter(obj => obj.status === 'fulfilled'))
    .then(results => results.map(result => result.value))
    .then(console.log);
```

allSettled()函式的作用，是透過區域函式 statusObject()個別地呼叫 Promise 的 then()方法，各個 Promise 若達成，就傳回 status 特性為'fufilled'的物件，若否決傳回 status 特性為'rejected'的物件❶，這些 Promise 被 Promise.all()收集起來，Promise.all()傳回的 Promise，就可以取得具有 status 特性的物件陣列❷，接下來就看是對達成的 Promise 或是被否決的 Promise 感興趣，這可以針對物件的 status 特性來進行判斷❸。

如果環境支援 ES11，有個 Promise.allSettled()[2]函式可以使用，就不需要如範例這樣自行定義了。

6.2.3 Promise 與產生器

在 3.3.6 談過產生器，當 Promise 與產生器結合時，可以產生有趣的操作風格，因為產生器函式中，yield 右邊的結果，會是產生器 next()傳回物件的 value 特性值，如果有個產生器是這麼寫的呢？

```
let generator = function*() {
    let r1 = yield asyncFoo(10);
    let r2 = yield asyncFoo(r1);
```

[2] Promise.allSettled：github.com/tc39/proposal-promise-allSettled

```
    let r3 = yield asyncFoo(r2);
    console.log(r3);
}();
```

在這邊，使用了 6.2.1 中 async-foo.js 的 asyncFoo()函式，也就是被 yield 的物件是 Promise 實例，因此，首次 generator.next()的話，得到的 Promise 會是 asyncFoo(10)的傳回值，如果進一步這麼撰寫呢？

```
let promise = generator.next();
promise.then(r => generator.next(r));
```

在 3.3.6 談過，可以在呼叫產生器的 next()方法時指定引數，令其成為 yield 的結果，因此上面的片段執行過後，r 值就會是產生器函式中 r1 值，接著若再執行底下流程：

```
promise = generator.next();
promise.then(r => generator.next(r));
```

流程就又會回到產生器函式中，r2 就會被指定任務達成的值...如果寫個 async()函式來執行這個重複不斷的過程，直到產生器函式執行完定義的流程，會是如何呢？

promise generator-promise.js

```
function asyncFoo(n) {
    return new Promise(resolve => {
        setTimeout(
            () => resolve(n * Math.random()),
            2000
        );
    });
}

function async(g) {
    let generator = g();

    function consume(result) {
        if(result.done) {
            return;
        }
        let promise = result.value;
        promise.then(r => consume(generator.next(r)));
    }

    consume(generator.next());
}

async(function*() {
    let r1 = yield asyncFoo(10);
```

```
    let r2 = yield asyncFoo(r1);
    let r3 = yield asyncFoo(r2);
    console.log(r3);
});
```

單看產生器函式中的三個 yield 片段，似乎循序的風格，事實上也是如此，四行程式碼確實是依序執行，跟一開始的這個風格比比看：

```
asyncFoo(10)
    .then(r1 => asyncFoo(r1))
    .then(r2 => asyncFoo(r2))
    .then(r3 => console.log(r3));
console.log('end');
```

上面這段程式碼，本質上還是非同步地，也就是說有可能先顯示了 end 字樣，任務才達成顯示 r3 的結果；然而，generator-promise.js 的撰寫風格除了極像是語法層面的支援之外，確實也只在任務達成後，才進行下一個任務。

實際上，若環境支援 ES8，新增的 async、await 就提供了語法層面的支援，而且執行上更有效率，這會是接下來 6.3 節要探討的主題。

6.3 async、await

Promise 解決了非同步任務回呼地獄的問題，然而是屬於 API 層面的方案，基本上還是需要搭配回呼函式，而且本質上還是非同步的，總是沒那麼方便；ES8 新增了 async、await，從語法層面上支援非同步任務的程式碼撰寫，更易於與同步任務的程式碼結合運用。

6.3.1 async 函式

就語義上，async 用來標示函式執行時是非同步，也就是函式中定義了獨立於程式主流程的任務，若呼叫 async 函式的話，會傳回一個 Promise 物件：

```
> async function foo() {
...     return 'foo';
... }
undefined
> foo()
Promise { 'foo' }
> foo().then(console.log)
Promise { <pending> }
> foo
```

```
[AsyncFunction: foo]
>
```

async 函式是 AsyncFunction 的實例，型態為 Function 的子類型，雖然函式中定義了 return 傳回字串'foo'，實際上這代表了 async 函式傳回的 Promise 達成任務後的結果，若要取得結果，方式之一就是使用 Promise 的 then()方法，另一個方式是使用稍後會介紹的 await 來取得。

async 函式中 return 的結果，若是 Promise 或 thenable 物件，可以取得該物件達成任務後的結果，因此可用來銜接其他傳回 Promise 的非同步函式。例如，底下範例會在 1 秒之後顯示數字的結果（而不是顯示 Promise 實例）：

async-await async-foo.js

```
function asyncFoo(n) {
    return new Promise(resolve => {
        setTimeout(
            () => resolve(n * Math.random()),
            1000
        );
    });
}

async function foo(n) {
    return asyncFoo(n);
}

foo(10).then(v => console.log(v));
```

如果 async 函式執行之後，無法達成任務呢？可以使用 **throw**，例如，來改寫一下 6.2.2 的 resolve-reject2.js：

async-await async-reject.js

```
async function randomDivided(divisor) {
    let n = Math.floor(Math.random() * 10);
    if(n !== 0) {
        return divisor / n;
    }
    throw new Error('Shit happens: divided by zero');
}

randomDivided(10)
    .then(
        n   => console.log(n),
        err => console.log(err)
    );
```

實際上，throw 是屬於 JavaScript 錯誤處理的語法，使用細節會在第 7 章說明；在 async 函式中若 throw，表示任務無法達成，也就可以使用 Promise 的 then()的第二個回呼函式來處理。

6.3.2 await 與 Promise

在 ES8 時，與 async 搭配的是 await，**就語義上，若想等待 async 函式執行完後，再執行後續的流程，可以使用 await**。例如：

```
async function foo() {
    return 'foo';
}

async function main() {
    let r = await foo();
    console.log(r); // 顯示 foo
}

main();
```

await 必須撰寫在 async 函式之中，若不想撰寫 main()函式，另一個方案是使用 3.3.4 談過的 IIFE：

```
async function foo() {
    return 'foo';
}

(async function() {
    let r = await foo();
    console.log(r);
})();
```

提示 »» 在撰寫本文的這個時間點，TC39 有個處於階段三的〈Top-level await[3]〉提案，可以在頂層直接撰寫 **await**。

實際上，**await 可以接上任何值，不一定要搭配 async 函式，只不過若是接上 Promise 實例，會在任務達成時，取得達成值作為 await 的結果**，因此 6.2.3 的 generator-promise.js，可以改寫為底下的範例：

[3] Top-level await：github.com/tc39/proposal-top-level-await

async-await async-foo2.js

```
function asyncFoo(n) {
    return new Promise(resolve => {
        setTimeout(
            () => resolve(n * Math.random()),
            2000
        );
    });
}

(async function() {
    let r1 = await asyncFoo(10);
    let r2 = await asyncFoo(r1);
    let r3 = await asyncFoo(r2);
    console.log(r3);
})();
```

await 時 Promise 若還沒達成任務，只是不會執行 async 函式中定義的後續流程，然而底層的事件迴圈並沒有被阻斷，若檢查到事件佇列中有新事件，還是會執行對應的任務；例如若在上例中加段程式碼：

```
…
setTimeout(
    () => console.log(n * Math.random()),
    1000
);
(async function() {
    let r1 = await asyncFoo(10);
    let r2 = await asyncFoo(r1);
    let r3 = await asyncFoo(r2);
    console.log(r3);
})();
```

在程式碼執行 await asyncFoo(10)後，需要 2 秒的時間才能取得任務達成值，然而 1 秒時間到時，setTimeout()設定的回呼函式還是會執行。

有些開發者會熟悉支援執行緒的語言，在 JavaScript 中會想尋找 sleep()之類、令執行緒停止指定時間的功能；然而，6.1.2 談過，JavaScript 不支援多執行緒，若令唯一的執行緒停止指定時間，結果會像在櫃台前排隊時，令櫃台人員發呆指定的時間，什麼事都不做，想想看隊伍後頭的人會怎樣？

然而基於方才的描述，await 時 Promise 若還沒達成任務，底層的事件迴圈並沒有被阻斷，因而透過 await 與 Promise，倒是可以模擬 sleep()的功能，例如底下的程式片段，就流程來說確實是間隔了一秒：

```
async function sleep(millis) {
    return new Promise(resolve => setTimeout(resolve, millis));
}

(async function() {
    console.log(new Date());
    await sleep(1000);
    console.log(new Date());
})();
```

並不是有了 async、await，就不需要 Promise 了，理由之一是 async 函式執行後也是傳回 Promise；另一個理由是在某些場合，必須使用 Promise 來銜接回呼風格的 API。例如，在 6.1.3 曾稍微提及 fs 模組的 readFile()函式，就可以自訂個 readTxtFile()如下傳回 Promise：

```
let fs = require('fs');

function readTxtFile(filename) {
    return new Promise(resolve => {
        fs.readFile(filename, function(_, content) {
            resolve(content.toString());
        });
    });
}

readTxtFile('foo.js')
    .then(console.log);
```

使用 Promise 來銜接回呼風格的 API 之後，進一步地就可以搭配 async、await 來使用了。例如：

```
(async function() {
    let content = await readTxtFile('foo.js');
    console.log(content);
})();
```

6.3.3 for-await-of 與非同步產生器函式

6.2.2 談過 Promise.all()，可以將一組 Promise 組合為一個 Promise，被指定的三個 Promise 都達成後，才算是 Promise.all()的結果達成；然而有時候，必須迭代處理一組 Promise，例如，處理 Promise 陣列：

```
function asyncFoo(n) {
    return new Promise(resolve => {
        setTimeout(
```

```
            () => resolve(n * Math.random()),
            2000
        );
    });
}

let promises = [asyncFoo(10), asyncFoo(20), asyncFoo(30)];
for(let p of promises) {
    p.then(console.log);
}
```

在上例中，for..of 每次迭代的元素是 Promise，因此不可以直接 console.log(p)，而必須透過 then()方法來顯示結果，不過要小心的是，這種寫法下，p.then(console.log)本身也是非同步，for..of 迴圈並不會等待目前 Promise 達成才迭代下一個，若需要滿足這個需求，必須使用 await 改為：

```
(async function() {
    let promises = [asyncFoo(10), asyncFoo(20), asyncFoo(30)];
    for(let p of promises) {
        console.log(await p);
    }
})();
```

ES9 新增了 **for-await-of** 語法，對於以上的需求，也可以直接撰寫為：

```
(async function() {
    let promises = [asyncFoo(10), asyncFoo(20), asyncFoo(30)];
    for await (let v of promises) {
        console.log(v);
    }
})();
```

必須迭代處理一組 Promise 的另一情境是，有個產生器函式，每次迭代時都會傳回 Promise：

```
function* asyncFoos(nums) {
    for(let n of nums) {
        yield asyncFoo(n);
    }
}

for(let p of asyncFoos([10, 20, 30])) {
    p.then(console.log);
}
```

同樣地，for..of 每次迭代的元素是 Promise，因此不可以直接 console.log(p)，而是必須透過 then()方法來顯示結果；for..of 迴圈也不會

等待目前的 Promise 達成才迭代下一個，若需要滿足這個需求，必須結合 await 改為：

```
(async function() {
    for(let p of asyncFoos([10, 20, 30])) {
        console.log(await p);
    }
})();
```

若支援 ES9 的 for-await-of 語法，也可以直接撰寫為：

```
(async function() {
    for await (let v of asyncFoos([10, 20, 30])) {
        console.log(v);
    }
})();
```

實際上，ES9 還支援**非同步產生器函式**，async 函式也可以是個產生器函式。例如：

async-await async-foo3.js

```
function asyncFoo(n) {
    return new Promise(resolve => {
        setTimeout(
            () => resolve(n * Math.random()),
            2000
        );
    });
}

async function* asyncFoos(nums) {
    for(let n of nums) {
        yield asyncFoo(n);
    }
}

(async function() {
    for await (let v of asyncFoos([10, 20, 30])) {
        console.log(v);
    }
})();
```

非同步產生器函式中，可以 yield 任何值，也可以是 Promise 或 thenable 物件，非同步產生器函式呼叫後傳回的物件，可以搭配 for-await-of 語法，等待每次任務達成後，取得值並執行迴圈本體。

課後練習

實作題

1. 請撰寫一個 toAsync()函式，可以接受同步函式，傳回一個新函式，新函式可以接受呼叫相同的引數，然而會是傳回 Promise 的非同步函式。例如：

```
function fib(n) {
    if(n === 0 || n === 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}

let asyncFib = toAsync(fib);
asyncFib(10).then(console.log); // 顯示 55
```